Taller Guiado - Análisis de Algoritmos

Alberto Luis Vigna Arroyo, Camilo José Martinez

7 de febrero de 2024

Abstract

En este documento se presenta el análisis del algoritmo de ordenamiento Bubble Sort, Insertion Sort y Quick Sort. Cada uno de estos algoritmos aborda el problema de ordenar una lista de elementos, pero difieren en sus enfoques y eficiencias. A continuación, se proporciona un resumen de los aspectos más destacados de cada algoritmo, destacando sus características, ventajas y desventajas, así como sus complejidades temporales y espaciales. Este análisis permite comprender mejor la idoneidad y el rendimiento de cada algoritmo en diferentes contextos y escenarios.

Part I
Análisis y diseño del problema

1 Análisis

El problema, informalmente, se puede describir como calcular para una secuencia de elementos $S = \langle a_1, a_2, a_3, \cdots, a_n \rangle$ en donde $\forall_i a_i \in \mathbb{T}$ y $\mathbb{T}$ existe una relación de orden parcial $' \leq'$, una secuencia ordenada, es decir una permutación $S = \langle a'_1, a'_2, a'_3, \cdots, a'_n \rangle$ en donde $a_i \leq a'_{i+1}$ y $a'_i \in S$.

2 Diseño

Con las observaciones presentadas en el análisis anterior, podemos escribir el diseño de un algoritmo que solucione el problema . A veces este diseño se conoce como el «contrato» del algoritmmos o las «precondiciones» y «poscondiciones» del algoritmo. El diseño se compone de entradas y salidas:

Definition. Entradas:

1. $S = \langle a_1, a_2, a_3, \cdots, a_n \rangle$, en donde $\forall$,a $\in \mathbb{T}$ y en $\mathbb{T}$ existe una relación de orden parcial $\leq$.

Definition. Salidas:

1. $S' = \langle a'_1, a'_2, a'_3, \cdots, a'_n \rangle$ en donde $a'_i \leq a'_{i+1}$ y $a'_i \in S$.

Part II
Algoritmos

3 Opción algoritmo 1 - BubbleSort

3.1 Algoritmo

Este algoritmo se basa en el principio de comparar pares de elementos adyacentes en una lista y realizar intercambios si es necesario, repitiendo este proceso hasta que la lista esté completamente ordenada. En cada iteración se va recorriendo la lista y verificando si el elementos $a_i > a_{i+1}$. Si este lo es los elementos se "rotan" para que estos vayan siendo ordenados.

En el algoritmo 1 se puede encontrar la implementación de este algoritmo.

Algorithm 1 BubbleSort

```
1:  procedure BubbleSort(S)
2:      n ← |S|
3:      for i ← 1 to n do
4:          for j ← 1 to n − i do
5:              if S[j] > S[j + 1] then
6:                  variableTemporal = S[j]
7:                  S[j] = S[j + 1]
8:                  S[j + 1] = variableTemporal
9:              end if
10:         end for
11:     end for
12: end procedure
```

3.2 Complejidad

El algoritmo Bubble Sort tiene órden de complejidad $O(n^2)$. Esto se debe a que emplea dos bucles anidados, donde el bucle externo realiza n iteraciones, y el bucle interno realiza comparaciones y posibles intercambios adyacentes. Esta estructura cuadrática resulta en un rendimiento menos eficiente, especialmente para conjuntos de datos extensos.

3.3 Invariante

En cada iteración del bucle externo, el elemento más grande entre los elementos no ordenados está ubicado en la última posición correcta de la lista.

- **Inicio**: Antes de iniciar el ordenamiento consideramos que la parte derecha de la lista (desde el índice i hasta n) contiene los elementos más grandes y ya está ordenada. Inicialmente, $i = 1$, por lo que la lista completa se toma como no ordenada.
- **Avance**: En cada iteración del bucle externo, el bucle interno compara y posiblemente intercambia elementos adyacentes. Esta operación mueve el elemento más grande a la posición $n - i$, manteniendo la parte derecha de la lista ordenada. El valor de i se incrementa después de cada iteración del bucle externo. Si no hay intercambios durante una iteración completa del bucle externo, significa que la lista está completamente ordenada y el algoritmo se detiene.
- **Terminación:** El proceso continua hasta que $i = n$, momentos en el cual se ha completado el ordenamiento de la lista.

3.4 Notas de implementación

Para la implementación de este algoritmo se hizo uso del lenguaje interpretado Python. En este caso, se nos brindó dicho código que fue escrito por el profesor Leonardo Florez El código para este algoritmo se ubica en la carpeta '$C\acute{o}digo/Taller1–An\acute{a}lisisdealgoritmos/sort/bubble_sort.py$'.

La implementación se encuentra entre la línea 11 y la 15.

4 Opción algoritmo 2 - Insertion Sort

4.1 Algoritmo

Este algoritmo se basa en el principio de iterar sobre una lista S de n elementos, donde $n = |S|$. La idea es que este algorimo empiece desde el segundo elemento ($i = 2$) y que en cada iteración se compare $S[i]$ con los elementos anteriores (es decir, $S[i-1], S[i-2], \ldots$), desplazándolos hacia la derecha mientras $S[i]$ sea menor. La meta es insertar $S[i]$ en la posición correcta, asegurando que la sublista $S[1:i]$ esté ordenada después de cada iteración.

En el algoritmo 2 se puede encontrar la implementación de este algoritmo.

4.2 Complejidad

La complejidad del algoritmos de Insertion Sort es de $O(n^2)$, lo que significa que el tiempo de ejecución crece cuadráticamente con el tamaño de la lista. En el peor caso, donde la lista está en orden descendente, cada elemento requiere

Algorithm 2 InsertionSort

```
1: procedure InsertionSort(S)
2:     n ← |S|
3:     for i ← 2 to n do
4:         key = S[i]
5:         j = i − 1
6:         while j > 0 and S[j] > key do
7:             S[j + 1] = S[j]
8:             j = j + 1
9:         end while
10:        S [j + 1] = key
11:    end for
12: end procedure
```

comparaciones y desplazamientos en cada iteración, resultando en aproximadamente $\frac{n^2}{2}$ operaciones. Aunque es eficiente para listas pequeñas o casi ordenadas, Insertion Sort puede volverse ineficiente para conjuntos de datos más grandes debido a su complejidad cuadrática.

4.3 Invariante

Después de cada iteración del bucle externo, la sublista que va desde el inicio de la lista hasta el índice actual $S[1 : i]$ está ordenada. Esto implica que los elementos en la sublista ya están en sus posiciones finales correctas en el contexto de la lista completa. El algoritmo, en cada paso, inserta el elemento actual en su posición correcta, desplazando los elementos mayores a la derecha para mantener la ordenación.

- **Inicio**: Antes de iniciar el bucle externo, la sublista inicial $S[1 : 1]$ consiste en un solo elemento, y por lo tanto, está trivialmente ordenada. El invariante se cumple inicialmente, ya que no hay elementos previos para comparar.

- **Avance**: Durante cada iteración del bucle externo, el algoritmo selecciona un nuevo elemento $S[i]$y lo compara con los elementos anteriores (es decir, $S\left[i-1\right], S\left[i-2\right], \ldots$). El bucle interno desplaza los elementos mayores hacia la derecha, creando espacio para insertar $S[i]$ en su posición correcta. Al finalizar la iteración, la sublista $S[1 : i]$ está ordenada, cumpliendo así el invariante. El avance garantiza que, después de cada iteración, la porción de la lista considerada hasta el momento sigue estando en orden.

- **Terminación:** Cuando el bucle externo ha recorrido toda la lista, la sublista completa $S[1 : n]$ está ordenada. El invariante se mantiene hasta el final del proceso, asegurando que, en cada paso del bucle externo, la sublista parcial se mantenga ordenada. Por lo tanto, al terminar, el invariante garantiza que la lista completa esté ordenada.

4.4 Notas de implementación

Para la implementación de este algoritmo se hizo uso del lenguaje interpretado Python. En este caso, se nos brindó dicho código que fue escrito por el profesor Leonardo Florez El código para este algoritmo se ubica en la carpeta '$Código/Taller1–Análisisdealgoritmos/sort/insertion_sort.py$'.

La implementación se encuentra entre la línea 11 y la 19.

5 Opción algoritmo 3 - Quick Sort

5.1 Algortimo

El algoritmo de QuickSort consiste en organizar una lista S. Inicialmente el algoritmo selecciona un elementos comúnmente llamado "*pivot*" de la lista, particionando los demás elementos en dos sublistas: aquella que contiene los elementos que son menores al pivot ($S_1 = \{x \in S | x < pivot\}$) y aquellos elementos que son mayores ($S_1 = \{x \in S | x > pivot\}$). Este enfoque divide la tarea en subproblemas más pequeños y se combina para lograr la ordenación completa. Se aplica el mismo proceso de manera recursiva a ambas sublistas, seleccionando nuevos pivotes y dividiéndolas hasta que toda la lista esté ordenada. El éxito de QuickSort radica en la elección eficiente del pivot y la partición de la lista en torno a él, reduciendo así el problema original en subproblemas más pequeños.

El paso clave es la elección del pivot y la partición de la lista en torno a él. Un pivot bien elegido mejora el rendimiento del algoritmo. QuickSort es eficiente en la práctica y a menudo supera a otros algoritmos de ordenamiento como Merge Sort debido a su menor número de intercambios.

En el algoritmo 3 se puede encontrar la implementación de este algoritmo.

5.2 Complejidad

Inicialmente tenemos que encontrar las complejidades de las funciones auxiliares. La función "particion" es de complejidad $O(n)$. Esta complejidad es posible encontrarla gracias al método de inspección. Por otro lado la función "intercambiar" es de complejidad $O(1)$.

Siguiendo con el análisis, para encontrar la complejidad de el algoritmo en general haremos uso de el teorema maestro.

5.2.1 Mejor Caso

Este algoritmo cuenta con la siguiente función de recurrencia:

$$T(n) = 2T(\frac{n}{2}) + n$$

Usando el teorema maestro se sigue el siguiente procedimiento:

- Encontrar el $C_{critico}$:

$$log_2 2 = 1$$

Algorithm 3 QuickSort

```
1: procedure QUICKSORT(S, inicio, fin)
2:     if inicio < fin then
3:         pivotIndex = particion(S, inicio, fin)
4:
5:         QuickSort(S, inicio, pivotIndex − 1)
6:         QuickSort(S, pivotIndex + 1, fin)
7:     end if
8: end procedure
9: procedure PARTICION(S, inicio, fin)
10:    pivot = S[fin]
11:    pivotIndex = inicio − 1
12:    for i ← inicio to fin − 1 do
13:        if S[i] ≤ pivot then
14:            pivotIndex = pivotIndex + 1
15:            intercambio(S, pivotIndex, i)
16:        end if
17:    end for
18:    intercambio(S, pivotIndex + 1, fin)
19:    return(pivotIndex + 1)
20: end procedure
21: procedure INTERCAMBIAR(S, i, j)
22:    temp = S[i]
23:    S[i] = S[j]
24:    S[j] = temp
25: end procedure
```

- Encontrar usando las fórmulas para este caso (Caso #2):

$$f(n) \in \theta(n^{log_2 2} log_2^0 n) \Longrightarrow f(n) \in \theta(n^{log_2 2} log_2^{0+1} n)$$

$$f(n) \in \theta(n) \Longrightarrow fn(n) \in \theta(n * log_2 n)$$

5.2.2 Peor Caso

Este peor caso se da cuando el algoritmo se divide en 2 subsecciónes en donde una es de tamaño $S' = 1$ y la otra es de tamaño $S' = n - 1$. Esto tendria que repetirse todas las veces, quedando subsecciones de tamaño $n-2, n-3, n-4, \ldots$

Teniendo en cuenta ese contexto podemos concluir que la función de recurrencia es la siguiente:

$$T(n) = 2T(n-1) + O(n) \Longrightarrow O(n^2)$$

$$T(n) = \frac{n * (n-1)}{2} \Longrightarrow n^2$$

5.3 Invariante

Al analizar el algoritmo de QuickSort, identificamos que el invariante crítico reside en el procedimiento de partición. Este procedimiento presenta un ciclo que define la ordenación de la secuencia.

El invariante fundamental en QuickSort establece que, después de cada operación de partición, el pivote ocupa su posición final en la lista ordenada. Formalmente, en la sublista $S'[1 : pivote - 1]$ son menores o iguales al pivote, y todos los elementos en la sublista a $S'[pivote + 1 : fin]$ son mayores o iguales al pivote. Entonces, el invariante puede expresarse como:

$$S'(1, pivotIndex - 1) < pivote < S'(pivotIndex + 1, fin)$$

Donde $pivotIndex$ es la posición final del pivote después de la partición. Este invariante garantiza que, en cada paso, la lista se divide en dos sublistas que cumplen con las condiciones de orden en relación con el pivote (los menores a la izquierda y los mayores a la derecha).

- **Inicio**: Al comenzar el algoritmo, antes de realizar cualquier partición, el invariante se establece. Sea $S[1 : n]$ la lista completa a ordenar, y i la posición del pivote antes de realizar la primera partición. En este momento, el invariante es: $P(0, i-1) \bigwedge P(i+1, n) \bigwedge a[i] = pivote$ Donde $P(0, i-1)$ y $P(i+1, n)$ indican que las sublistas a la izquierda y a la derecha del pivote, respectivamente, están inicialmente vacías.

- **Avance**: Después de realizar una partición en la posición i, el invariante se mantiene. Se elige un pivote y se reorganizan los elementos de la lista de modo que los elementos a la izquierda del pivote sean menores o iguales, y los elementos a la derecha sean mayores o iguales. El invariante después del avance es: $P(0, i-1) \bigwedge P(i+1, n) \bigwedge a[i] = pivote$ Donde $P(0, i-1)$ y

$P(i+1, n)$ indican que las sublistas a la izquierda y a la derecha del pivote, respectivamente, cumplen con las condiciones de orden establecidas por el pivote después de la partición.

- **Terminación:** Cuando el algoritmo termina, todas las particiones han sido realizadas y la lista está completamente ordenada. En este punto, el invariante final es:

  $$P(1, n)$$

  Esto significa que toda la lista cumple con las condiciones de orden, ya que la lista se ha dividido y ordenado correctamente en cada partición.

5.4 Notas de Implementación

Para la implementación de este algoritmo se hizo uso del lenguaje interpretado Python. En este caso, se nos brindó dicho código que fue escrito por el profesor Leonardo Flórez El código para este algoritmo se ubica en la carpeta '*Código/Taller*1–*Análisisdealgoritmos/sort/quicksort.py*'.

La implementación se encuentra entre la línea 12 y la 22.

Parte III
Comparación de los algoritmos

6 Ejecución de los algoritmos:

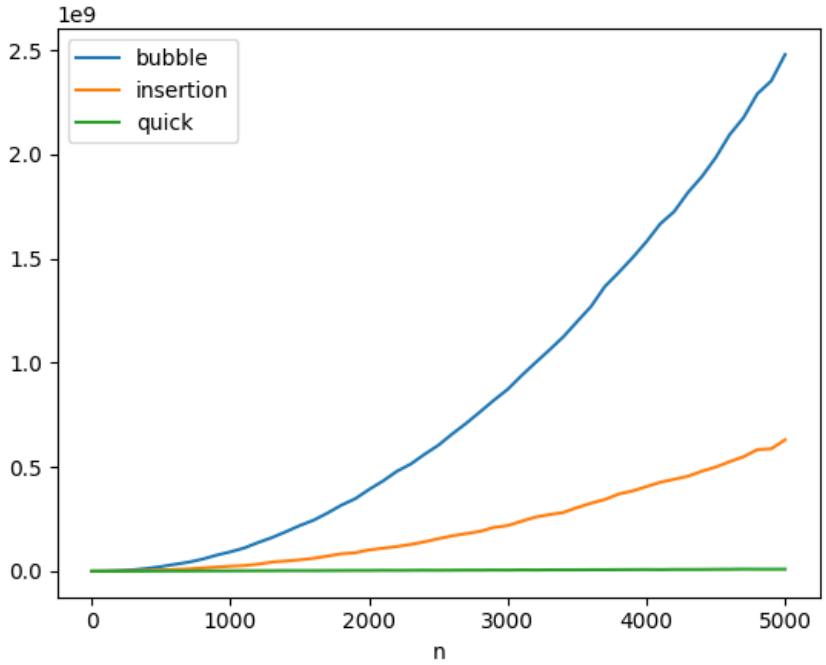
Para poder realizar estas comparaciones se corrió el programa "main.py". En este programa podemos comparar los 3 tipos de algortimos y de esta manera, poder realizar una comparación con ayuda de gráficas y los mismos outputs que nos brinda el código. Para poder realizar las pruebas hemos creado un archivo llamado **data.res** con los siguientes argumentos:

$$min\_s = 1$$

$$max\_s = 5000$$

$$step = 100$$

Con esos argumentos fuimos capaces de obtener la siguiente gráfica:



En la tabla anterior se puede apreciar de manera interpretativa que para el algoritmo de Bubble Sort, Insertion Sort & Quick Sort. Estos se comportan de manera similar a su complejidad algoritmica en el peor caso; siendo $O(n^2)$ para BubbleSort, $O(n^2)$ para el InsertionSort y $O(n * log(n))$ para el QuickSort.

Ahora, podemos ver que aunque ambos Bubble Sort e Insertion Sort tienen una complejidad asintótica cuadrática ($O(n^2)$), la diferencia clave en su rendimiento radica en cómo manejan el intercambio de elementos. **En Bubble Sort, se realizan intercambios continuos entre elementos adyacentes hasta que todo el conjunto esté ordenado**,lo que puede resultar en un alto número de operaciones incluso en conjuntos parcialmente ordenados. Por otro lado, **Insertion Sort construye gradualmente la parte ordenada del conjunto, desplazando elementos solo cuando es necesario para insertar el siguiente elemento en su posición correcta.** Esto implica que Insertion Sort tiende a ser más eficiente, especialmente en conjuntos de datos parcialmente ordenados, ya que puede detenerse temprano en cada iteración si encuentra que un elemento ya está en su lugar correcto, lo que reduce el número total de comparaciones e intercambios en comparación con Bubble Sort.

6.1 BubbleSort

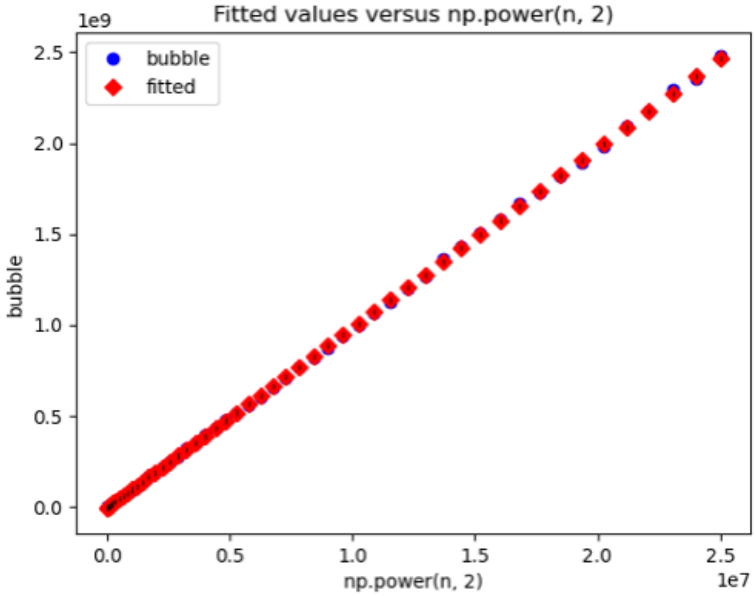
OLS Regression Results

Dep. Variable:	bubble	R-squared:	1.000
Model:	OLS	Adj. R-squared:	1.000
Method:	Least Squares	F-statistic:	4.752e+05
Date:	Mon, 05 Feb 2024	Prob (F-statistic):	2.39e-99
Time:	15:36:08	Log-Likelihood:	-880.63
No. Observations:	51	AIC:	1765.
Df Residuals:	49	BIC:	1769.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-5.609e+06	1.63e+06	-3.449	0.001	-8.88e+06	-2.34e+06
np.power(n, 2)	98.7692	0.143	689.379	0.000	98.481	99.057

Omnibus:	2.676	Durbin-Watson:	1.634
Prob(Omnibus):	0.262	Jarque-Bera (JB):	1.877
Skew:	0.452	Prob(JB):	0.391
Kurtosis:	3.260	Cond. No.	1.69e+07

Con un $R^2 = 1$, podemos afirmar que el modelo de la regresión se ajusta a n^2 y que la variabilidad es explicada por las variables del modelo. El modelo explica toda la variabilidad en la variable dependiente.



Ya con la gráfica podemos ver la similitud de la simulación con la función n^2.

6.2 InsertionSort

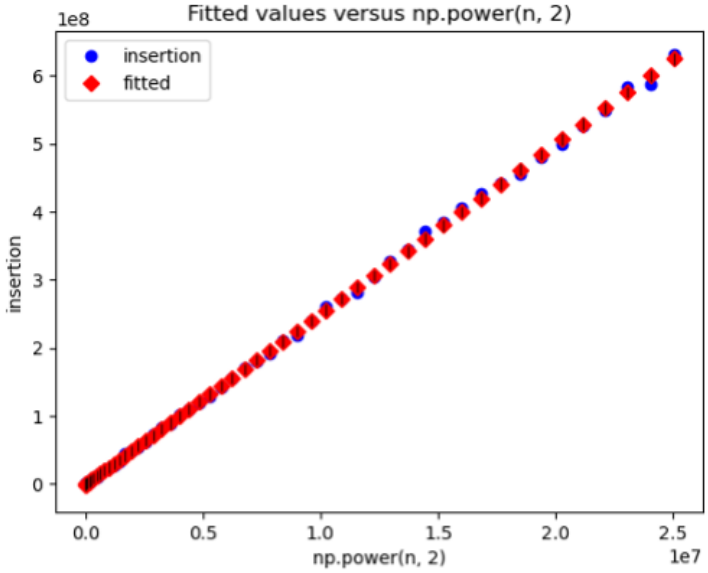
OLS Regression Results

Dep. Variable:	insertion	R-squared:	1.000
Model:	OLS	Adj. R-squared:	1.000
Method:	Least Squares	F-statistic:	1.193e+05
Date:	Mon, 05 Feb 2024	Prob (F-statistic):	1.21e-84
Time:	15:36:10	Log-Likelihood:	-845.81
No. Observations:	51	AIC:	1696.
Df Residuals:	49	BIC:	1699.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-6.522e+05	8.22e+05	-0.794	0.431	-2.3e+06	9.99e+05
np.power(n, 2)	25.0005	0.072	345.372	0.000	24.855	25.146

Omnibus:	4.716	Durbin-Watson:	1.831
Prob(Omnibus):	0.095	Jarque-Bera (JB):	5.286
Skew:	-0.173	Prob(JB):	0.0712
Kurtosis:	4.539	Cond. No.	1.69e+07

Con un $R^2 = 1$, podemos afirmar que el modelo de la regresión se ajusta a n^2 y que la variabilidad es explicada por las variables del modelo. El modelo explica toda la variabilidad en la variable dependiente.



De manera similar al Bubble Sort, vemos la concordancia de la simulación con la función n^2. Para este caso se ve un poco más de variación, pero esta es insignificante.

6.3 QuickSort

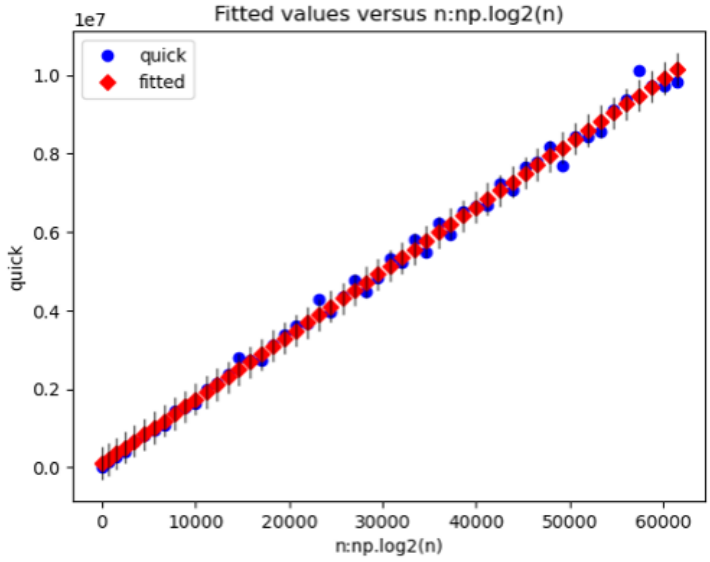
OLS Regression Results

Dep. Variable:	quick	R-squared:	0.996
Model:	OLS	Adj. R-squared:	0.996
Method:	Least Squares	F-statistic:	1.164e+04
Date:	Mon, 05 Feb 2024	Prob (F-statistic):	6.41e-60
Time:	15:36:09	Log-Likelihood:	-693.77
No. Observations:	51	AIC:	1392.
Df Residuals:	49	BIC:	1395.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	1.028e+05	5.21e+04	1.974	0.054	-1837.316	2.07e+05
n:np.log2(n)	163.6222	1.517	107.880	0.000	160.574	166.670

Omnibus:	5.758	Durbin-Watson:	2.354
Prob(Omnibus):	0.056	Jarque-Bera (JB):	5.281
Skew:	0.478	Prob(JB):	0.0713
Kurtosis:	4.254	Cond. No.	6.39e+04

Con un $R^2 = 0.996$, podemos afirmar que el modelo de la regresión se ajusta de manera extremadamente precisa a $n * log(n)$ y que la variabilidad es explicada en gran parte por las variables del modelo. El modelo explica casi toda la variabilidad en la variable dependiente.



En este caso, se observa mayor variabilidad pequeña pero suficiente para que $R^2 \neq 1.00$. De igual manera, vemos que esta no altera el comportamiento del experimento con el $n * log(n)$ en la regresión, por lo cual se puede explicar el $R^2 = 0.996$ el cual es bastante preciso.